Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de sistemas e industrial
Curso: Ingeniería de Software 1 (2016701)

Testing Unitario TheBigBro:

●Tomás Nicolás Jerez García.
●Jacobo Ulises Cortés Duque.
●Santiago Zamora Garzón.
●Gabriel Mateo Gonzalez Lara.

Introducción:
En este documento representamos 12 pruebas unitarias implementadas para validar el funcionamiento esencial de la aplicación **LogiTrace** que es una aplicación web con la cual se administra stocks y gestión de productos.

Herramientas utilizadas:

Componente	**Uso**
Framework	pytest y pytest-django.
Lenguaje	python 3.11.9
Tipo de prueba	Prueba unitaria
Comandos de ejecución	python -m pytest

Implementación de las pruebas:
Cada integrante del grupo implemento 3 pruebas que cumplían con las características pedidas para la entrega, las pruebas implementadas son las siguientes:

Test para validar si un correo es interno corporativo:

```python
def test_validar_correo_operador_interno(self):
        assert is_internal_email("planner@logitrace.co")
        assert not is_internal_email("externo@gmail.com")
```

Funcionalidad Validada:

- Verificación de la regla de negocio que exige correos con el dominio corporativo **@logitrace.co**.
- Identificación y exclusión de correos personales o provenientes de dominios externos.

Casos Límite:

- Correos vacíos o nulos.
- Correos con dominios no corporativos (por ejemplo, *gmail.com*, *hotmail.com*).
- Direcciones con formato incorrecto o incompleto.
- Variaciones en mayúsculas y minúsculas en el dominio.
- Dominios similares pero no exactamente iguales (por ejemplo, *logitrace.com*, *logitrace.co.externo.com*).

Importancia:
Asegura que únicamente los usuarios pertenecientes a la organización sean identificados como internos, lo cual es esencial para mantener controles de acceso, flujos internos seguros y una correcta segmentación de usuarios dentro del sistema.

Test Validación de campos para el producto:

```python
def test_validar_campos_requeridos_producto(self):
        payload = {
            "sku": "SKU-001",
            "name": "Palet",
            "category": "standard",
            "reorder_point": 5,
        }
        require_fields(payload, ["sku", "name", "category"])
        with pytest.raises(ValueError):
```

```
            require_fields(payload, ["sku", "name",
"description"])
```

Funcionalidades validadas:

- Identificación que debe tener cada producto **["sku", "name", "category"]** con el cual se podrá registrarlo.
- Análisis de campos del producto con payload para comprobar su correcta colocación.

Casos limites:

- No hay concordancia entre el payload y los campos colocados por el administrador .
- Algún campo vacío.
- Producto no registrado de la manera correcta.

Importancia:

El registro del producto es fundamental para tener los objetos para poder organizar y que tenga el flujo básico de la aplicación, si no llegamos a tener productos bien registrados , el usuario no podrá hacer ninguna transacción con el.

Test de validación de despacho:

```python
def test_validar_ventana_despacho(self):
        start = datetime(2024, 3, 10, 9, 0)
        end = start + timedelta(hours=3)
        validate_dispatch_window(start, end)
        with pytest.raises(ValueError):
            validate_dispatch_window(start, start)
```

Funcionalidad Validada:
Verificación de la regla de negocio que exige que las ventanas de despacho tengan una duración válida y lógica, donde la fecha/hora de fin debe ser posterior a la fecha/hora de inicio.

Identificación y exclusión de ventanas temporales inválidas:

- Ventanas con fecha de fin igual a la de inicio
- Ventanas con fecha de fin anterior a la de inicio
- Ventanas con duración cero o negativa

Casos Límite:

- Ventanas con la misma fecha y hora de inicio y fin
- Ventanas donde el fin es anterior al inicio
- Ventanas con duración mínima (fracciones de segundo)
- Ventanas extremadamente largas (posiblemente con límites máximos)
- Fechas nulas o en formato incorrecto
- Ventanas que cruzan límites de días, meses o años

Importancia:
Asegura que los horarios de despacho programados en el sistema sean temporalmente coherentes y evita la creación de ventanas de tiempo imposibles o ilógicas. Esto es fundamental para la planificación logística, la asignación de recursos y la prevención de errores en la programación de entregas, manteniendo la integridad temporal de las operaciones de despacho.

Test para priorizar envío:

```python
def test_prioridades_envio(self):
        validate_priority_tag("low")
        validate_priority_tag("high")
        with pytest.raises(ValueError):
            validate_priority_tag("urgent")
```

Funcionalidad Validada:

- Priorización de unos tags que elija el cliente.
- Exclusión de prioridades a otros estados que no queramos en el momento.

Casos Límites:

- Prioridades vacías o nulas.
- Palabras que no tienen nada que ver con un estado .
- Variaciones en las mayúsculas y minúsculas del estado.

Importancia:

Esto es fundamental cuando tenemos unos productos que se necesitan entregar en un tiempo más corto y enfocarse en ellos, esto para tener una mayor eficacia y poder organizar de la mejor manera la energía y el tiempo.

Test notificación válida:

```python
    def test_notificacion_valida(self):
         payload = NotificationPayload(
             event_type="inventory_alert",
             message="Ubicación LOC-A supera el 95% de
ocupación",
             recipients=[1, 2],
         )
         assert payload.is_valid()
         assert "inventory_alert" in ALLOWED_NOTIFICATION_TYPES
```

Funcionalidad válida:

- Hace un registro Payload. En el cual tiene una notificación sobre alguna novedad que haya en alguno de los recipientes.
- Que el dominio tiene tipos de eventos limitados y controlados.
- Que la notificación que se envía pase toda las validaciones para el envío.

Casos limites:

- Algún campo está vacío.

- Que en mensaje no tenga algún tipo de destinatario.
- Notificaciones que no tienen algún sentido y son rechazadas.

Importancia: Al momento de querer hacer un tipo de reporte para enviarlo para su cambio, es fundamental tener coherencia en todos los aspecto que se quieren dar , de esta manera aseguraremos la eficacia de los reportes y que adicional se tiene una verificación de dicha información para su debida aprobación.

Test para el cálculo de eficiencia de entregas:

Python

```python
def test_calculo_fill_rate(self):
        assert calculate_fill_rate(8, 10) == 80.0
        assert calculate_fill_rate(12, 10) == 100.0
        with pytest.raises(ValueError):
            calculate_fill_rate(0, 0)
```

Funcionalidad válida:

- Calcula la eficiencia que tiene LogiTrace con todas las entregas que entrega de las totales que realiza.
- Calcula en un porcentaje te 0 al 100% .
- Algún cálculo llega a dar más de 100% , el sistema lo aproxima directamente a 100%.

Casos limites:

- Datos que no sean números.
- Datos que estén en 0 , ya que estos no proporcionan alguna información al sistema.

Importancia:

El cálculo de cuánta eficiencia tiene Logic Trace con cada pedido exitoso del total ayudará a determinar si toda la parte interna está funcionando correctamente o si está bajo . Buscar la razón exacta del porqué está mal .

Test para validación de transiciones de pedido:

```python
def test_validar_transiciones_pedido(self):
        validate_order_status_transition(OrderStatus.CREATED,
OrderStatus.RESERVED)
        validate_order_status_transition(OrderStatus.RESERVED,
OrderStatus.DISPATCHED)
        with pytest.raises(ValueError):

validate_order_status_transition(OrderStatus.CREATED,
OrderStatus.DISPATCHED)
```

Funcionalidad Validada:

- Verificación de la regla de negocio que define qué transiciones de estado de un pedido son válidas dentro del flujo operativo.
- Confirmación de que las transiciones permitidas, como **CREATED** → **RESERVED** y **RESERVED** → **DISPATCHED**, se ejecuten sin generar errores.
- Detección de transiciones no permitidas, como **CREATED** → **DISPATCHED**, asegurando que el sistema lance una excepción cuando el flujo se intenta saltar pasos obligatorios.

Casos Límite:

- Intentos de saltar etapas intermedias, provocando una transición inválida.
- Estados desconocidos o no reconocidos dentro del flujo definido.
- Transiciones hacia el mismo estado (por ejemplo, **CREATED** → **CREATED**), dependiendo de la política del sistema.
- Transiciones en orden inverso (por ejemplo, **DISPATCHED** → **RESERVED**), si están prohibidas por el flujo del negocio.

Importancia:
Garantiza que el ciclo de vida del pedido siga estrictamente el flujo establecido por la operación, evitando inconsistencias, saltos indebidos y estados imposibles. Esto asegura la integridad del proceso logístico y previene errores que podrían afectar inventarios, entregas o reportes operativos.

Test para el código de referencia por ubicación:

```python
def test_codigo_referencia_ubicacion(self):
        assert is_valid_reference_code("LOC-BX-0001")
        assert not is_valid_reference_code("loc-1")
```

Funcionalidad Validada:

- Verificación de la regla de negocio que define el formato válido para códigos de referencia de ubicación, asegurando que cumplan un patrón estandarizado, como **“LOC-BX-0001”**.
- Detección de códigos que no cumplen dicho formato, evitando valores incompletos, informales o no estructurados.

Casos Límite:

- Códigos con estructura incompleta o con menos segmentos de los requeridos.
- Uso de minúsculas o combinaciones inconsistentes de mayúsculas y minúsculas, dependiendo de si el sistema exige un formato estricto.
- Códigos con numeración inválida o con caracteres no permitidos.
- Códigos demasiado cortos o excesivamente largos respecto al estándar definido.
- Valores nulos o cadenas vacías.

Importancia:
Asegura que cada ubicación del sistema esté identificada mediante un código uniforme, legible y procesable, evitando ambigüedades y errores en inventario, almacenamiento o trazabilidad. Esto contribuye al orden y a la coherencia de los datos operativos.

Test para validar la capacidad proyectada:

```python
def test_validar_capacidad_proyectada(self):
        total = validate_capacity_projection(100, 70, 20,
safety_margin=5)
        assert total == 95
        with pytest.raises(ValueError):
            validate_capacity_projection(80, 60, 30)
```

Funcionalidad Validada:

- Verificación del cálculo correcto de la capacidad proyectada a partir de la capacidad total, la capacidad utilizada, la capacidad comprometida y el margen de seguridad.
- Confirmación de que el resultado final sea coherente y que se apliquen correctamente reglas como restar las capacidades consumidas y sumar el margen de seguridad.
- Validación de que se lance una excepción cuando la proyección excede la capacidad disponible, evitando valores imposibles o incoherentes.

Casos Límite:

- Situaciones donde la suma de la capacidad utilizada y comprometida supera la capacidad total.
- Valores negativos o inconsistentes para cualquiera de los parámetros.
- Falta de margen de seguridad explícito, usando el valor por defecto del sistema.
- Casos donde el cálculo exacto resulta en cero capacidad restante o en un margen extremadamente reducido.

- Entradas muy grandes o muy pequeñas que puedan provocar errores aritméticos o validaciones incorrectas.

Importancia:
Garantiza que los cálculos operativos relacionados con la capacidad sean precisos y seguros, evitando sobrecargas, asignaciones excesivas o decisiones basadas en información errónea. Esto protege la integridad de la planificación operativa y permite tomar decisiones confiables sobre disponibilidad y carga de trabajo.

Test para validar cantidad positiva:

```python
def test_validar_cantidad_positiva(self):
        validate_positive_quantity(10)
        with pytest.raises(ValueError):
            validate_positive_quantity(0)
```

Funcionalidad Validada:

- Verificación de que la función acepte únicamente cantidades estrictamente positivas, permitiendo valores mayores a cero como en el caso de 10.
- Confirmación de que se genere una excepción cuando la cantidad no cumple el requisito mínimo, como sucede con 0, asegurando que no se procesen valores inválidos.

Casos Límite:

- Cantidades iguales a cero, que deben ser rechazadas según la regla de negocio.
- Valores negativos, que también deberían provocar una excepción.
- Cantidades extremadamente grandes que podrían requerir validaciones adicionales dependiendo del contexto del sistema.
- Tipos de datos no numéricos o valores nulos, si la función debe manejarlos explícitamente.

Importancia:
Garantiza que las operaciones que dependen de cantidades válidas no se vean afectadas por valores incorrectos o no permitidos. Esto evita errores en cálculos, movimientos de

inventario, asignaciones o cualquier proceso que requiera una cantidad mínima válida para operar correctamente.

Test para ver el sanitizado de los comentarios

```python
def test_sanitizar_comentario(self):
        assert sanitize_comment("  Ajuste   de    inventario
") == "Ajuste de inventario"
        assert sanitize_comment("") == ""
```

Funcionalidad Validada:

- Verificación de que la función elimine espacios innecesarios al inicio, en medio y al final del comentario, normalizando el texto a un formato limpio y consistente.
- Confirmación de que un comentario vacío sea procesado correctamente y devuelto tal cual, sin generar errores ni valores inesperados.

Casos Límite:

- Comentarios con múltiples espacios consecutivos entre palabras.
- Comentarios que contienen solo espacios o caracteres en blanco.
- Cadenas vacías o nulas, dependiendo de lo que defina la función.
- Uso de caracteres especiales o saltos de línea que podrían requerir normalización adicional.
- Comentarios muy largos que podrían poner a prueba el rendimiento o el manejo de memoria.

Importancia:
Asegura que los comentarios ingresados por los usuarios o generados automáticamente

mantengan un formato uniforme, evitando inconsistencias en reportes, auditorías o logs. Esto mejora la legibilidad, la calidad de los datos y previene problemas derivados de información mal formateada en los sistemas operativos o administrativos.

Test para validar roles permitidos:

```python
def test_validar_roles_permitidos(self):
        for rol in ALLOWED_ROLES:
            validate_role(rol)
        with pytest.raises(ValueError):
            validate_role("guest")
```

Funcionalidad Validada:

- Verificación de que la función acepte únicamente los roles definidos como permitidos en la constante **ALLOWED_ROLES**, asegurando que todos los valores autorizados sean reconocidos como válidos.
- Confirmación de que cualquier rol no incluido en la lista preestablecida, como el caso de **"guest"**, genere una excepción, evitando asignaciones indebidas o accesos no autorizados.

Casos Límite:

- Roles que no existan en la lista de permitidos pero que puedan parecer válidos por su nombre.
- Roles escritos con mayúsculas, minúsculas o combinaciones que requieran normalización previa, dependiendo de la implementación.
- Intentos de validar valores nulos, vacíos o tipos de dato que no correspondan a un rol.
- Escenarios donde la lista de roles permitidos esté vacía o haya sido modificada erróneamente.

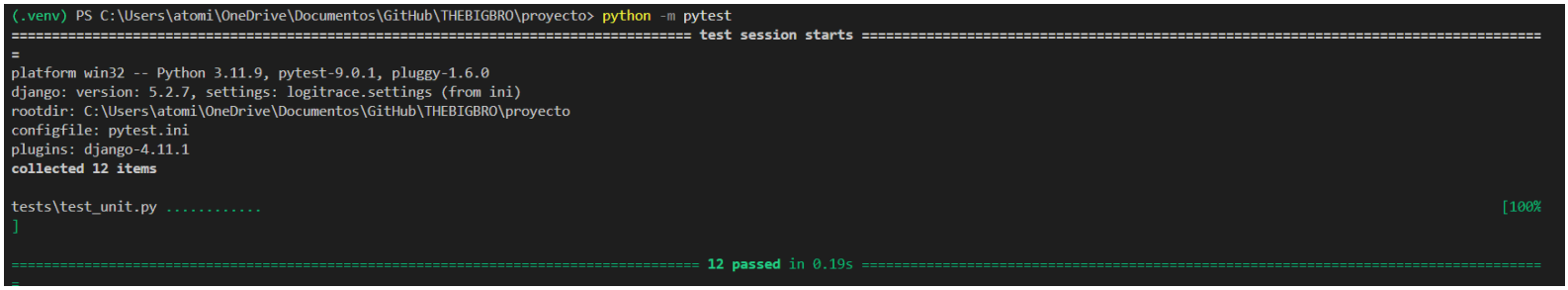
Importancia:
Asegura que el sistema solo procesa y reconoce roles legítimos, protegiendo los controles de acceso y evitando que usuarios sin permisos adecuados ejecuten acciones restringidas. Estas validaciones fortalecen la seguridad, la integridad del flujo de permisos y la correcta segmentación de funcionalidades dentro del sistema.

Ejecución de pruebas:

Para la ejecución de todos los test, corrimos en nuestro entorno virtual en la dirección de **THEBIGBRO\proyecto** el siguiente comando:

```
Shell
THEBIGBRO\proyecto> python -m pytest
```

Al ejecutarlo nos, los 12 test que teníamos fueron efectuados y nos dio la siguiente salida:




```
Shell
(.venv) PS C:\Users\atomi\OneDrive\Documentos\GitHub\THEBIGBRO\proyecto>
python -m pytest
============================================================================
======== test session starts
============================================================================
=========
platform win32 -- Python 3.11.9, pytest-9.0.1, pluggy-1.6.0
django: version: 5.2.7, settings: logitrace.settings (from ini)
rootdir: C:\Users\atomi\OneDrive\Documentos\GitHub\THEBIGBRO\proyecto
configfile: pytest.ini
```

```
plugins: django-4.11.1
collected 12 items

tests\test_unit.py ............
[100%]

================================================================================
========= 12 passed in 0.19s
================================================================================
=========
```

Los resultados muestran que la totalidad de las pruebas unitarias ejecutadas se completaron satisfactoriamente, con 12 tests aprobados y sin errores ni fallos. Esto indica que las funciones validadas, están funcionando conforme a las reglas de negocio definidas. Además, el tiempo de ejecución reducido refleja un conjunto de pruebas ligero y eficiente ya que su ejecución fue en el tiempo deseado < 1 segundo. En conjunto, estos resultados proporcionan un nivel de confianza alto en la estabilidad y consistencia de la lógica central del sistema, al menos en el ámbito cubierto por las pruebas actuales.